Społeczna Akademia Nauk w Łodzi

Programowanie współbieżne

Laboratorium 3

Prowadzący
mgr inż. Krystian Gumiński

2025

Tworzenie nowego wątku jest kosztowne

W programowaniu współbieżnym w Javie jednym z pierwszych rozwiązań, które przychodzi do głowy, jest stworzenie nowego wątku (Thread) dla każdego zadania, które chcemy wykonać. Podejście to wydaje się intuicyjne, ale w praktyce okazuje się nieefektywne i trudne w utrzymaniu przy większej liczbie zadań.

Każdy wątek w systemie operacyjnym wymaga określonej ilości zasobów: pamięci - m.in. na stos (zwykle kilka megabajtów na wątek), czasu procesora: utworzenie nowego wątku to operacja systemowa, konfiguracji: system musi przydzielić wątkowi kontekst wykonania, stos, identyfikator, rejestry i inne struktury danych.

Jeśli aplikacja uruchamia setki lub tysiące zadań, ciągłe tworzenie i niszczenie wątków prowadzi do: zwiększonego obciążenia systemu, fragmentacji pamięci, spowolnienia działania programu z powodu przełączania kontekstu (ang. context switching).

Kiedy w systemie działa wiele wątków, procesor musi co pewien czas przełączać się między nimi. Każde przełączenie oznacza: zapisanie aktualnego stanu wątku (rejestrów, licznika programu), odczytanie stanu innego wątku, wznowienie jego wykonywania.

To działanie jest kosztowne. Nie wykonuje się w tym czasie faktycznego kodu programu, a jedynie administracyjne operacje systemowe. Im więcej wątków, tym częstsze i droższe przełączanie, co może prowadzić do paradoksalnej sytuacji, w której więcej wątków = wolniejsze działanie programu. Żeby temu zapobiec stworzono pule wątków. Tworzymy pewną pule wątków, które czekają na zadania. Już są utworzone i są w gotowość do uruchomienia. Przykład na listningu 1

Listing 1: Pula wątków

```
1  ExecutorService executor = Executors.newFixedThreadPool(3);
2
3  for (int i = 1; i <= 5; i++) {
4      final int taskId = i;
5      executor.submit(() -> {
6          System.out.println("Zadanie " + taskId + " wykonuje " +
               Thread.currentThread().getName());
7      try {
8              Thread.sleep(1000);
9      } catch (InterruptedException e) {
10         Thread.currentThread().interrupt();
11     }});
12 }
13
14 executor.shutdown();
```

Listing 2: Wynik listningu 1

```
1 Zadanie 1 wykonuje pool-1-thread-1
2 Zadanie 3 wykonuje pool-1-thread-3
3 Zadanie 2 wykonuje pool-1-thread-2
```

```
Zadanie 4 wykonuje pool-1-thread-3
Zadanie 5 wykonuje pool-1-thread-1
```

Na listningu 3 Tworzony jest licznik typu AtomicInteger, który może być bezpiecznie modyfikowany przez wiele wątków jednocześnie. Służy on do zliczania liczby zadań, które nie zostały przyjęte przez pulę wątków (czyli odrzuconych). Dzięki temu nie ma potrzeby stosowania synchronizacji - operacje getAndIncrement() są atomowe (nieprzerywalne i bezpieczne współbieżnie).

Omówienie parametrów:

- corePoolSize = 1 minimalna liczba aktywnych wątków w puli
- maximumPoolSize = 4 maksymalna liczba wątków, które mogą być aktywne jednocześnie
- keepAliveTime = 30s czas, po którym nadmiarowe wątki (powyżej 1) zostaną usunięte, jeśli są bezczynne
- LinkedBlockingQueue<>(10) kolejka o długości 10 - przechowuje zadania czekające na wykonanie
- (r, e) -> ... niestandardowy RejectedExecutionHandler, czyli strategia obsługi zadań, które nie mogą zostać przyjęte

Metoda (r, e) zostanie wywołana za każdym razem, gdy zadanie zostanie odrzucone przez executor. Do odrzucenia dochodzi, gdy w puli działa maksymalna liczba wątków (maximumPoolSize), kolejka jest pełna (10 zadań), a kolejne zadanie próbuje zostać dodane. Wtedy wykonuje się kod wewnątrz handlera.

Listing 3: Pula wątków

```
AtomicInteger counter = new AtomicInteger(0);

Executor poolExecutor = new ThreadPoolExecutor(1, 4,
    30, TimeUnit.SECONDS, new LinkedBlockingQueue<>(10),
    (r, e) -> {
        if (counter.getAndIncrement() == 10) {
            System.out.println("10 tasks run unsuccessful, dear
                poolexecutor it is time to die");
            e.shutdown();
    }});

Runnable runnable = () -> {
    System.out.println(Thread.currentThread().getName());
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
```

```
17      }
18  };
19
20  IntStream.range(0, 30).forEach(i -> poolExecutor.execute(runnable)
        );
```

Listing 4: Wynik listningu 3

```
1  pool-2-thread-1
2  pool-2-thread-2
3  pool-2-thread-3
4  10 tasks run unsuccessful, dear poolexecutor it is time to die
5  pool-2-thread-4
6  pool-2-thread-1
7  pool-2-thread-2
8  pool-2-thread-3
9  pool-2-thread-4
10 pool-2-thread-1
11 pool-2-thread-2
12 pool-2-thread-3
13 pool-2-thread-4
14 pool-2-thread-1
15 pool-2-thread-2
```

ForkJoinPool

ForkJoinPool to specjalna implementacja puli wątków (ang. thread pool), zaprojektowana do efektywnego wykonywania zadań rekurencyjnych, które można podzielić na mniejsze części. Jej głównym celem jest maksymalne wykorzystanie wielordzeniowych procesorów w zadaniach, które można rozbić na wiele mniejszych podzadań.

Podstawowy model działania ForkJoinPool opiera się na rekurencyjnym podziale problemu na mniejsze fragmenty, aż do momentu, gdy każdy fragment można wykonać bez dalszego dzielenia.

- Duży problem: podziel na mniejsze części (fork).
- Każdy fragment wykonaj równolegle.
- Po zakończeniu łącz wyniki częściowych obliczeń (join).

Nazwa klasy ForkJoinPool pochodzi od dwóch kluczowych operacji wykonywanych w jej modelu działania - fork, czyli podziału zadania na mniejsze części, oraz join, czyli łączenia wyników cząstkowych w całość. Struktura ForkJoinPool opiera się na konkretnej hierarchii klas. Na najwyższym poziomie znajduje się klasa ForkJoinPool, która zarządza wszystkimi wątkami i zadaniami w systemie. Poniżej umieszczona jest klasa ForkJoinTask, stanowiąca

abstrakcyjną reprezentację zadania, które może zostać podzielone na mniejsze podzadania. Z klasy tej wywodzą się dwa typy zadań: RecursiveAction oraz RecursiveTask. Klasa RecursiveAction służy do definiowania zadań, które nie zwracają wyniku (np. operacje wykonujące określone działania), natomiast RecursiveTask reprezentuje zadania, które po zakończeniu zwracają wynik obliczeń.

Listing 5: Sumowanie tablicy

```
1
2     static class SumTask extends RecursiveTask<Integer> {
3         private static final int THRESHOLD = 3;
4         private final int[] array;
5         private final int start;
6         private final int end;
7
8         public SumTask(int[] array, int start, int end) {
9             this.array = array;
10            this.start = start;
11            this.end = end;
12        }
13
14        @Override
15        protected Integer compute() {
16            if (end - start <= THRESHOLD) {
17                int sum = 0;
18                for (int i = start; i < end; i++) {
19                    sum += array[i];
20                }
21                System.out.println(Thread.currentThread().getName
                      () + " liczy fragment od " + start + " do " +
                      (end - 1));
22                return sum;
23            } else {
24                int mid = (start + end) / 2;
25                SumTask left = new SumTask(array, start, mid);
26                SumTask right = new SumTask(array, mid, end);
27
28                left.fork();
29                int rightResult = right.compute();
30                int leftResult = left.join();
31
32                return leftResult + rightResult;
33            }
34        }
35    }
```

```
36
37      public static void main(String[] args) {
38          int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
39
40          ForkJoinPool pool = new ForkJoinPool();
41          SumTask task = new SumTask(numbers, 0, numbers.length);
42
43          int result = pool.invoke(task);
44          System.out.println("Suma wszystkich elementow = " + result
                );
45      }
```

Listing 6: Wynik listningu 5

```
1 ForkJoinPool-1-worker-1 liczy fragment od 6 do 8
2 ForkJoinPool-1-worker-3 liczy fragment od 0 do 1
3 ForkJoinPool-1-worker-4 liczy fragment od 4 do 5
4 ForkJoinPool-1-worker-2 liczy fragment od 2 do 3
5 Suma wszystkich elementow = 45
```

- THRESHOLD określa, kiedy zadanie jest wystarczająco małe, by nie dzielić go dalej.
- fork(), uruchamia podzadanie asynchronicznie.
- compute(), wykonuje zadanie lokalnie (rekurencyjnie).
- join(), czeka na wynik podzadania.
- ForkJoinPool automatycznie rozdziela pracę między dostępne wątki.

1 Zadanie

W załączniku do zadania znajduje się archiwum z plikami tekstowymi. Napisać program, który zlicza łączną liczbę słów znajdujących się w wszystkich plikach tekstowych, korzystając z równoległego przetwarzania.

Zadaniem programu jest:

- Przejść rekurencyjnie przez folder.
- Dla każdego pliku .txt policzyć liczbę słów.
- Połączyć wyniki i zwrócić łączną liczbę słów we wszystkich plikach.
- Zrobić to równolegle

Wymóg 1.1. Zadanie zwróć w postaci sprawozdania z umieszczonymi screenami wykonania programu, oddzielnego pliku z kodem orazem oddzielnym plikiem txt, który zawiera log policznoych wyrazów w formie:
ForkJoinPool-1-worker-13 -> file14949.txt: 7819 słów
ForkJoinPool-1-worker-14 -> file21808.txt: 8827 słów
ForkJoinPool-1-worker-10 -> file5946.txt: 8137 słów
...
Łączna liczba słów: X